A C++17 Thread Pool for High-Performance Scientific Computing

Barak Shoshany*

Department of Physics, Brock University,
1812 Sir Isaac Brock Way, St. Catharines, Ontario, L2S 3A1, Canada

Abstract

We present a modern C++17-compatible thread pool implementation, built from scratch with high-performance scientific computing in mind. The thread pool is implemented as a single lightweight and self-contained class, and does not have any dependencies other than the C++17 standard library, thus allowing a great degree of portability. In particular, our implementation does not utilize OpenMP or any other high-level multithreading APIs, and thus gives the programmer precise low-level control over the details of the parallelization, which permits more robust optimizations. The thread pool was extensively tested on both AMD and Intel CPUs with up to 40 cores and 80 threads. This paper provides motivation, detailed usage instructions, performance tests, and the full annotated source code[1].

Contents

*E-mail address: bshoshany@brocku.ca, website: https://baraksh.com/

[1] The source code is freely available for download on GitHub: https://github.com/bshoshany/thread-pool

1 Introduction

1.1 Motivation

Multithreading[1] is essential for modern high-performance computing. Since C++11, the C++[2][3][4] standard library has included built-in low-level multithreading support using constructs such as `std::thread`. However, `std::thread` creates a new thread each time it is called, which can have a significant performance overhead. Furthermore, it is possible to create more threads than the hardware can handle simultaneously, potentially resulting in a substantial slowdown.

In this paper we present a modern C++17-compatible[5] thread pool implementation, the class `thread_pool`, which avoids these issues by creating a fixed pool of threads once and for all, and then reusing the same threads to perform different tasks throughout the lifetime of the pool. By default, the number of threads in the pool is equal to the maximum number of threads that the hardware can run in parallel.

The user submits tasks to be executed into a queue. Whenever a thread becomes available, it pops a task from the queue and executes it. Each task is automatically assigned an `std::future`, which can be used to wait for the task to finish executing and/or obtain its eventual return value.

In addition to `std::thread`, the C++ standard library also offers the higher-level construct `std::async`, which may internally utilize a thread pool - but this is not guaranteed, and in fact, currently only the MSVC implementation[6] of `std::async` uses a thread pool, while GCC and Clang do not. Using our custom-made thread pool class instead of `std::async` allows the user more control, transparency, and portability.

High-level multithreading APIs, such as OpenMP[7], allow simple one-line automatic parallelization of C++ code, but they do not give the user precise low-level control over the details of the parallelization. The thread pool class presented here allows the programmer to perform and manage the parallelization at the lowest level, and thus permits more robust optimizations, which can be used to achieve considerably higher performance.

As demonstrated in section 5, using our thread pool class we were able to saturate the upper bound of expected speedup for matrix multiplication and generation of random matrices. These performance tests were performed on 12-core / 24-thread and 40-core / 80-thread systems using GCC on Linux.

1.2 Overview of features

- **Fast:**
  - Built from scratch with maximum performance in mind.
  - Suitable for use in high-performance computing clusters with a very large number of CPU cores.
  - Compact code, to reduce both compilation time and binary size.
  - Reusing threads avoids the overhead of creating and destroying them for individual tasks.
  - A task queue ensures that there are never more threads running in parallel than allowed by the hardware.
- **Lightweight:**
  - Only ~160 lines of code, excluding comments, blank lines, and the two optional helper classes.
  - Single header file: simply `#include "thread_pool.hpp"`.
  - Header-only: no need to install or build the library.

- Self-contained: no external requirements or dependencies. Does not require OpenMP or any other multithreading APIs. Only uses the C++ standard library, and works with any C++17-compliant compiler.
- **Easy to use:**
  - Very simple operation, using a handful of member functions.
  - Every task submitted to the queue automatically generates an `std::future`, which can be used to wait for the task to finish executing and/or obtain its eventual return value.
  - Optionally, tasks may also be submitted without generating a future, sacrificing convenience for greater performance.
  - The code is thoroughly documented using Doxygen comments - not only the interface, but also the implementation, in case the user would like to make modifications.
- **Additional features:**
  - Automatically parallelize a loop into any number of parallel tasks.
  - Easily wait for all tasks in the queue to complete.
  - Change the number of threads in the pool safely as needed.
  - Fine-tune the sleep duration of each thread's worker function for optimal performance.
  - Synchronize output to a stream from multiple threads in parallel using the `synced_stream` helper class.
  - Easily measure execution time for benchmarking purposes using the `timer` helper class.

2 Basic usage

2.1 Including the library

To use the thread pool library, simply include the header file:

```
#include "thread_pool.hpp"
```

The thread pool will now be accessible via the `thread_pool` class.

2.2 Constructors

The default constructor creates a thread pool with as many threads as the hardware can handle concurrently, as reported by the implementation via `std::thread::hardware_concurrency()`. With a hyperthreaded CPU, this will be twice the number of CPU cores. This is probably the constructor you want to use. For example:

```
// Constructs a thread pool with as many threads as available in the hardware.
thread_pool pool;
```

Optionally, a number of threads different from the hardware concurrency can be specified as an argument to the constructor. However, note that adding more threads than the hardware can handle will **not** improve performance, and in fact will most likely hinder it. This option exists in order to allow using **less** threads than the hardware concurrency, in cases where you wish to leave some threads available for other processes. For example:

```
// Constructs a thread pool with only 12 threads.
thread_pool pool(12);
```

If your program's main thread only submits tasks to the thread pool and waits for them to finish, and does not perform any computationally intensive tasks on its own, then it is recommended to use the default value for the number of threads. This ensures that all of the threads available in the hardware will be put to work while the main thread waits.

However, if your main thread does perform computationally intensive tasks on its own, then it is recommended to use the value `std::thread::hardware_concurrency() - 1` for the number of threads. In this case, the main thread plus the thread pool will together take up exactly all the threads available in the hardware.

2.3 Submitting tasks to the queue

A task can be any function, with zero or more arguments, and with or without a return value. Once a task has been submitted to the queue, it will be executed as soon as a thread becomes available. Tasks are executed in the order that they were submitted (first-in, first-out).

The member function `submit()` is used to submit tasks to the queue. The first argument is the function to execute, and the rest of the arguments are the arguments to pass to the function, if any. The return value is an `std::future` associated to the task. For example:

```
// Submit a task without arguments to the queue, and get a future for it.
auto my_future = pool.submit(task);
// Submit a task with one argument to the queue, and get a future for it.
auto my_future = pool.submit(task, arg);
// Submit a task with two arguments to the queue, and get a future for it.
auto my_future = pool.submit(task, arg1, arg2);
```

Using `auto` for the return value of `submit()` is recommended, since it means the compiler will automatically detect which instance of the template `std::future` to use. The value of the future depends on whether the function has a return value or not:

- If the submitted function has a return value, then the future will be set to that value when the function finishes its execution.
- If the submitted function does not have a return value, then the future will be a `bool` that will be set to `true` when the function finishes its execution.

To wait until the future's value becomes available, use the member function `wait()`. To obtain the value itself, use the member function `get()`, which will also automatically wait for the future if it's not ready yet. For example:

```
// Submit a task and get a future.
auto my_future = pool.submit(task);
// Do some other stuff while the task is executing.
do_stuff();
// Get the task's return value from the future,
// waiting for it to finish running if needed.
auto my_return_value = my_future.get();
```

2.4 Parallelizing loops

Consider the following loop:

```
for (T i = start; i <= end; i++)
    loop(i);
```

where:

- T is any signed or unsigned integer type.
- `start` is the first index to loop over (inclusive).
- end is the last index to loop over (inclusive).
- `loop()` is a function that takes exactly one argument, the loop index, and has no return value.

This loop may be automatically parallelized and submitted to the thread pool's queue using the member function `parallelize_loop()` as follows:

```cpp
// Equivalent to the above loop, but will be automatically parallelized.
pool.parallelize_loop(start, end, loop);
```

The loop will be parallelized into a number of tasks equal to the number of threads in the pool, with each task executing the function `loop()` for a roughly equal number of indices. The main thread will then wait until all tasks generated by `parallelize_loop()` finish executing (and only those tasks - not any other tasks that also happen to be in the queue).

If desired, the number of parallel tasks may be manually specified using a fourth argument:

```cpp
// Parallelize the loop into 12 parallel tasks
pool.parallelize_loop(start, end, loop, 12);
```

For best performance, it is recommended to do your own benchmarks to find the optimal number of tasks for each loop (you can use the `timer` helper class - see section 4.2). Using less tasks than there are threads may be preferred if you are also running other tasks in parallel. Using more tasks than there are threads may improve performance in some cases.

2.5 Compiling and compatibility

This library should successfully compile on any C++17 standard-compliant compiler, on all operating systems for which such a compiler is available. Compatibility was verified with a 12-core / 24-thread AMD Ryzen 9 3900X CPU at 3.8 GHz using the following compilers and platforms:

- GCC v10.2.0 on Windows 10 build 19042.928.
- GCC v10.3.0 on Ubuntu 21.04.
- Clang v11.0.0 on Windows 10 build 19042.928 and Ubuntu 21.04.
- MSVC v14.28.29910 on Windows 10 build 19042.928.

In addition, this library was tested on a Compute Canada node equipped with two 20-core / 40-thread Intel Xeon Gold 6148 CPUs at 2.4 GHz, for a total of 40 cores and 80 threads, running CentOS Linux 7.6.1810, using the following compilers:

- GCC v9.2.0
- Intel C++ Compiler (ICC) v19.1.3.304

As this library requires C++17 features, the code must be compiled with C++17 support. For GCC, Clang, and ICC, use the `-std=c++17` flag. For MSVC, use `/std:c++17`. On Linux, you will also need to use the `-pthread` flag with GCC, Clang, or ICC to enable the POSIX threads library.

3 Advanced usage

3.1 Getting and resetting the number of threads in the pool

The member function `get_thread_count()` returns the number of threads in the pool. This will be equal to `std::thread::hardware_concurrency()` if the default constructor was used.

It is generally unnecessary to change the number of threads in the pool after it has been created, since the whole point of a thread pool is that you only create the threads once. However, if needed, this can be done - safely - using the `reset()` member function, which waits for all submitted tasks to be completed, then destroys all of the threads and creates a new thread pool with the desired new number of threads, as specified in the function's argument. If no argument is given to `reset()`, the new number of threads will be the hardware concurrency.

3.2 Submitting tasks to the queue without futures

Usually, it is best to submit a task to the queue using `submit()`. This allows you to wait for the task to finish and/or get its return value later. However, sometimes a future is not needed, for example when you just want to "set and forget" a certain task, or if the task already communicates with the main thread or with other tasks without using futures, such as via references or pointers. In such cases, you may wish to avoid the overhead involved in assigning a future to the task in order to increase performance.

The member function `push_task()` allows you to submit a task to the queue without generating a future for it. The task can have any number of arguments, but it cannot have a return value. For example:

```
// Submit a task without arguments or return value to the queue.
pool.push_task(task);
// Submit a task with one argument and no return value to the queue.
pool.push_task(task, arg);
// Submit a task with two arguments and no return value to the queue.
pool.push_task(task, arg1, arg2);
```

3.3 Manually waiting for all tasks to complete

To wait for a **single** submitted task to complete, use `submit()` and then use the `wait()` or `get()` member functions of the obtained future. However, in cases where you need to wait until **all** submitted tasks finish their execution, or if the tasks have been submitted without futures using `push_task()`, you can use the member function `wait_for_tasks()`.

Consider, for example, the following code:

```
thread_pool pool;
size_t a[100];
for (size_t i = 0; i < 100; i++)
    pool.push_task([&a, i] { a[i] = i * i; });
std::cout << a[50];
```

The output will most likely be garbage, since the task that modifies `a[50]` has not yet finished executing by the time we try to access that element (in fact, that task is probably still waiting in the queue). One solution would be to use `submit()` instead of `push_task()`, but perhaps we don't want the overhead of generating 100 different futures. Instead, simply adding the line

```
pool.wait_for_tasks();
```

after the `for` loop will ensure - as efficiently as possible - that all tasks have finished running before we attempt to access any elements of the array a, and the code will print out the value 2500 as expected. (Note, however, that `wait_for_tasks()` will wait for **all** the tasks in the queue, including those that are unrelated to the `for` loop. Using `parallelize_loop()` would make much more sense in this particular case, as it will wait only for the tasks related to the loop.)

3.4 Setting the worker function's sleep duration

The **worker function** is the function that controls the execution of tasks by each thread. It loops continuously, and with each iteration of the loop, checks if there are any tasks in the queue. If it finds a task, it pops it out of the queue and executes it. If it does not find a task, it will wait for a bit, by calling `std::this_thread::sleep_for()`, and then check the queue again. The public member variable `sleep_duration` controls the duration, in microseconds, that the worker function sleeps for when it cannot find a task in the queue.

The default value of `sleep_duration` is 1000 microseconds, or 1 millisecond. In our benchmarks, lower values resulted in high CPU usage when the workers were idle. The value of 1000 microseconds was roughly the minimum value needed to reduce the idle CPU usage to a negligible amount.

In addition, in our benchmarks this value resulted in moderately improved performance compared to lower values, since the workers check the queue - which is a costly process - less frequently. On the other hand, increasing the value even more could potentially cause the workers to spend too much time sleeping and not pick up tasks from the queue quickly enough, so 1000 is the "sweet spot".

However, please note that this value is likely unique to the particular system our benchmarks were performed on, and your own optimal value would depend on factors such as your OS and C++ implementation, the type, complexity, and average duration of the tasks submitted to the pool, and whether there are any other programs running at the same time. Therefore, it is strongly recommended to do your own benchmarks and find the value that works best for you.

If `sleep_duration` is set to 0, then the worker function will execute `std::this_thread::yield()` instead of sleeping if there are no tasks in the queue. This will suggest to the OS that it should put this thread on hold and allow other threads to run instead. However, this also causes the worker functions to have high CPU usage when idle. On the other hand, for some applications this setting may provide better performance than sleeping - again, do your own benchmarks and find what works best for you.

4 Optional helper classes

4.1 Synchronizing printing to an output stream

When printing to an output stream from multiple threads in parallel, the output may become garbled. For example, consider this code:

```
thread_pool pool;
for (auto i = 1; i <= 5; i++)
    pool.push_task([i] {
        std::cout << "Task no. " << i << " executing.\n";
    });
```

The output may look as follows:

```
Task no. Task no. 2Task no. 5 executing.
Task no.  executing.
Task no. 1 executing.
4 executing.
3 executing.
```

The reason is that, although each **individual** insertion to `std::cout` is thread-safe, there is no mechanism in place to ensure subsequent insertions from the same thread are printed contiguously.

The helper class `synced_stream` is designed to eliminate such synchronization issues. The constructor takes one optional argument, specifying the output stream to print to. If no argument is supplied, `std::cout` will be used:

```
// Construct a synced stream that will print to std::cout.
synced_stream sync_out;
// Construct a synced stream that will print to the output stream my_stream.
synced_stream sync_out(my_stream);
```

The member function `print()` takes an arbitrary number of arguments, which are inserted into the stream one by one, in the order they were given. `println()` does the same, but also prints a newline character `\n` at the end, for convenience. A mutex is used to synchronize this process, so that any other calls to `print()` or `println()` using the same `synced_stream` object must wait until the previous call has finished.

As an example, this code:

```
synced_stream sync_out;
thread_pool pool;
```

```
for (auto i = 1; i <= 5; i++)
    pool.push_task([i, &sync_out] {
        sync_out.println("Task no. ", i, " executing.");
    });
```

Will print out:

```
Task no. 1 executing.
Task no. 2 executing.
Task no. 3 executing.
Task no. 4 executing.
Task no. 5 executing.
```

4.2 Measuring execution time

If you are using a thread pool, then your code is most likely performance-critical. Achieving maximum performance requires performing a considerable amount of benchmarking to determine the optimal settings and algorithms. Therefore, it is important to be able to measure the execution time of various computations under different conditions. In the context of the thread pool class, you would probably be interested in finding the optimal number of threads in the pool and the optimal sleep duration for the worker functions.

The helper class `timer` provides a simple way to measure execution time. It is very straightforward to use:

1. Create a new `timer` object.
2. Immediately before you execute the computation that you want to time, call the `start()` member function.
3. Immediately after the computation ends, call the `stop()` member function.
4. Use the member function `ms()` to obtain the elapsed time for the computation in milliseconds.

For example:

```
timer tmr;
tmr.start();
do_something();
tmr.stop();
std::cout << "The elapsed time was " << tmr.ms() << " ms.\n";
```

5 Performance tests

To benchmark the performance of our thread pool class, we measured the execution time of various parallelized operations on large matrices, using a custom-built matrix class template. The test code makes use of version 1.3 of the `thread_pool` class (which is the most recent version at the time of writing) and implements a generalization of its `parallelize_loop()` member function adapted specifically for parallelizing matrix operations. Execution time was measured using the `timer` helper class.

For each matrix operation, we parallelized the computation into blocks. Each block consists of a number of atomic operations equal to the block size, and was submitted as a separate task to the thread pool's queue, such that the number of blocks equals the total number of tasks. We tested 6 different block sizes for each operation in order to compare their execution time.

5.1 AMD Ryzen 9 3900X (24 threads)

The first test was performed on a computer equipped with a 12-core / 24-thread AMD Ryzen 9 3900X CPU at 3.8 GHz, compiled using GCC v10.3.0 on Ubuntu 21.04 with the `-O3` compiler flag. The thread pool consisted of 24 threads, making full use of the CPU's hyperthreading capabilities.

The output of our test program was as follows:

```
Adding two 4800x4800 matrices:
With block size of 23040000 ( 1 block ), execution took 37 ms.
With block size of  3840000 ( 6 blocks), execution took 17 ms.
With block size of  1920000 (12 blocks), execution took 16 ms.
With block size of   960000 (24 blocks), execution took 17 ms.
With block size of   480000 (48 blocks), execution took 17 ms.
With block size of   240000 (96 blocks), execution took 17 ms.

Generating random 4800x4800 matrix:
With block size of 23040000 ( 1 block ), execution took 291 ms.
With block size of  3840000 ( 6 blocks), execution took 52 ms.
With block size of  1920000 (12 blocks), execution took 27 ms.
With block size of   960000 (24 blocks), execution took 25 ms.
With block size of   480000 (48 blocks), execution took 20 ms.
With block size of   240000 (96 blocks), execution took 17 ms.

Transposing one 4800x4800 matrix:
With block size of 23040000 ( 1 block ), execution took 129 ms.
With block size of  3840000 ( 6 blocks), execution took 24 ms.
With block size of  1920000 (12 blocks), execution took 19 ms.
With block size of   960000 (24 blocks), execution took 17 ms.
With block size of   480000 (48 blocks), execution took 16 ms.
With block size of   240000 (96 blocks), execution took 15 ms.

Multiplying two 800x800 matrices:
With block size of   640000 ( 1 block ), execution took 431 ms.
With block size of   106666 ( 6 blocks), execution took 88 ms.
With block size of    53333 (12 blocks), execution took 61 ms.
With block size of    26666 (24 blocks), execution took 42 ms.
With block size of    13333 (48 blocks), execution took 37 ms.
With block size of     6666 (96 blocks), execution took 32 ms.
```

In this test, we find a speedup by roughly a factor of 2 for addition, 9 for transposition, 13 for matrix multiplication, and 17 for random matrix generation. Here are some lessons we can learn from these results:

- For simple element-wise operations such as addition, multithreading improves performance very modestly, only by a factor of 2, even when utilizing every available hardware thread. This is because compiler optimizations already parallelize simple loops fairly well on their own. Omitting the `-O3` optimization flag, we observed a factor of 9 speedup for addition. However, the user will most likely be compiling with optimizations turned on anyway.
- Matrix multiplication and random matrix generation, which are more complicated operations that cannot be automatically parallelized by compiler optimizations, gain the most out of multithreading - with a very significant speedup by a factor of 15 on average. Given that the test CPU only has 12 physical cores, and hyperthreading can generally produce no more than a 30% performance improvement[8], a factor of 15 speedup is about as good as can be expected.
- Transposition also enjoys a factor of 9 speedup with multithreading. Note that transposition requires reading memory is non-sequential order, jumping between the rows of the source matrix, which is why, compared to sequential operations such as addition, it is much slower when single-threaded, but benefits more from multithreading, especially when split into smaller blocks.
- Even though the test CPU only has 24 threads, there is still a small but consistent benefit to dividing the computation into 48 or even 96 parallel blocks. This is especially significant in multiplication, where we get roughly a 25% speedup with 96 blocks (4 blocks per thread) compared to 24 blocks (1 block per thread).

5.2 Dual Intel Xeon Gold 6148 (80 threads)

The second test was performed on a Compute Canada node equipped with dual 20-core / 40-thread Intel Xeon Gold 6148 CPUs at 2.4 GHz, for a total of 40 cores and 80 threads, compiled using GCC v9.2.0 on CentOS Linux 7.6.1810 with the -O3 compiler flag. The thread pool consisted of 80 threads, making full use of the hyperthreading capabilities of both CPUs.

We adjusted the block sizes compared to the previous test, to match the larger number of threads. The output of our test program was as follows:

```
Adding two 4800x4800 matrices:
With block size of 23040000 (  1 block ), execution took 73 ms.
With block size of  1152000 ( 20 blocks), execution took 9 ms.
With block size of   576000 ( 40 blocks), execution took 7 ms.
With block size of   288000 ( 80 blocks), execution took 7 ms.
With block size of   144000 (160 blocks), execution took 8 ms.
With block size of    72000 (320 blocks), execution took 10 ms.

Generating random 4800x4800 matrix:
With block size of 23040000 (  1 block ), execution took 423 ms.
With block size of  1152000 ( 20 blocks), execution took 29 ms.
With block size of   576000 ( 40 blocks), execution took 15 ms.
With block size of   288000 ( 80 blocks), execution took 13 ms.
With block size of   144000 (160 blocks), execution took 11 ms.
With block size of    72000 (320 blocks), execution took 10 ms.

Transposing one 4800x4800 matrix:
With block size of 23040000 (  1 block ), execution took 167 ms.
With block size of  1152000 ( 20 blocks), execution took 18 ms.
With block size of   576000 ( 40 blocks), execution took 11 ms.
With block size of   288000 ( 80 blocks), execution took 9 ms.
With block size of   144000 (160 blocks), execution took 10 ms.
With block size of    72000 (320 blocks), execution took 12 ms.

Multiplying two 800x800 matrices:
With block size of   640000 (  1 block ), execution took 771 ms.
With block size of    32000 ( 20 blocks), execution took 57 ms.
With block size of    16000 ( 40 blocks), execution took 24 ms.
With block size of     8000 ( 80 blocks), execution took 21 ms.
With block size of     4000 (160 blocks), execution took 17 ms.
With block size of     2000 (320 blocks), execution took 15 ms.
```

In this test, we find a speedup by roughly a factor of 10 for addition, 19 for transposition, 42 for random matrix generation, and 51 for matrix multiplication. The last result again matches the estimation of a 30% improvement in performance due to hyperthreading, which indicates that we are once again saturating the maximum possible performance of our system.

An interesting point to notice is that for **single-threaded** calculations (1 block), the dual Xeon CPUs actually perform worse by up to a factor of 2 compared to the single Ryzen CPU. This is due to the base clock speed of the Ryzen (3.8 GHz) being considerably higher than the base clock speed of the Xeon (2.4 GHz). Since each core of the Xeon is slower than each core of the Ryzen, we need more parallelization to achieve the same overall speed. However, with full parallelization (24 threads on the Ryzen, 80 threads on the Xeon), the latter is faster by about a factor of 2.

6 Annotated source code

The complete source code for the header file `thread_pool.hpp` is provided below, including the thread pool class and the two helper classes. We have included detailed Doxygen-style comments within the code to explain the purpose and usage of each member function and variable.

The code should not be copied directly from the PDF; it is only included here for documentation purposes, as conversion to PDF format will most likely make the code unusable. To download the source code, please visit the GitHub repository: https://github.com/bshoshany/thread-pool

Note that the code presented here is from version 1.3 of the library, which was the most recent at the time of writing, but may have been superseded by a newer version since this paper went to press.

6.1 Preprocessor directives

The following preprocessor directives are placed at the top of the file `thread_pool.hpp`. The comments indicate which elements of the C++17 standard library are used from each header file.

```
#pragma once

#include <algorithm>   // std::max
#include <atomic>      // std::atomic
#include <chrono>      // std::chrono
#include <cstdint>     // std::int_fast64_t, std::uint_fast32_t
#include <functional>  // std::function
#include <future>      // std::future, std::promise
#include <iostream>    // std::cout, std::ostream
#include <memory>      // std::shared_ptr, std::unique_ptr
#include <mutex>       // std::mutex, std::scoped_lock
#include <queue>       // std::queue
#include <thread>      // std::this_thread, std::thread
#include <type_traits> // std::decay_t, std::enable_if_t,
                       // std::is_void_v, std::invoke_result_t
#include <utility>     // std::move, std::swap
```

6.2 The `thread_pool` class

This section contains the complete source code for the `thread_pool` class.

```
class thread_pool
{
    typedef std::uint_fast32_t ui32;

public:
    // ============================
    // Constructors and destructors
    // ============================

    /**
     * @brief Construct a new thread pool.
     *
     * @param _thread_count The number of threads to use. The default value is
     * the total number of hardware threads available, as reported by the
     * implementation. With a hyperthreaded CPU, this will be twice the number
     * of CPU cores. If the argument is zero, the default value will be used
     * instead.
```

```cpp
     */
    thread_pool(const ui32 &_thread_count = std::thread::hardware_concurrency())
        : thread_count(_thread_count ?
            _thread_count : std::thread::hardware_concurrency()),
        threads(new std::thread[_thread_count ?
            _thread_count : std::thread::hardware_concurrency()])
    {
        create_threads();
    }

    /**
     * @brief Destruct the thread pool. Waits for all submitted tasks to be
     * completed, then destroys all threads.
     */
    ~thread_pool()
    {
        wait_for_tasks();
        running = false;
        destroy_threads();
    }

    // =======================
    // Public member functions
    // =======================

    /**
     * @brief Get the number of threads in the pool.
     *
     * @return The number of threads.
     */
    ui32 get_thread_count() const
    {
        return thread_count;
    }

    /**
     * @brief Parallelize a loop by splitting it into blocks, submitting each
     * block separately to the thread pool, and waiting for all blocks to finish
     * executing. The loop will be equivalent to: for (T i = first_index; i <=
     * last_index; i++) loop(i);
     *
     * @tparam T The type of the loop index. Should be a signed or unsigned
     * integer.
     * @tparam F The type of the function to loop through.
     * @param first_index The first index in the loop (inclusive).
     * @param last_index The last index in the loop (inclusive).
     * @param loop The function to loop through. Should take exactly one
     * argument, the loop index.
     * @param num_tasks The maximum number of tasks to split the loop into. The
     * default is to use the number of threads in the pool.
     */
    template <typename T, typename F>
    void parallelize_loop(T first_index, T last_index,
```

```
        const F &loop, ui32 num_tasks = 0)
    {
        if (num_tasks == 0)
            num_tasks = thread_count;
        if (last_index < first_index)
            std::swap(last_index, first_index);
        size_t total_size = last_index - first_index + 1;
        size_t block_size = total_size / num_tasks;
        if (block_size == 0)
        {
            block_size = 1;
            num_tasks = std::max((ui32)1, (ui32)total_size);
        }
        std::atomic<ui32> blocks_running = 0;
        for (ui32 t = 0; t < num_tasks; t++)
        {
            T start = (T)(t * block_size + first_index);
            T end = (t == num_tasks - 1) ? last_index :
                (T)((t + 1) * block_size + first_index - 1);
            blocks_running++;
            push_task([start, end, &loop, &blocks_running] {
                for (T i = start; i <= end; i++)
                    loop(i);
                blocks_running--;
            });
            while (blocks_running != 0)
            {
                std::this_thread::yield();
            }
        }
    }

    /**
     * @brief Push a function with no arguments or return value into the task
     * queue.
     *
     * @tparam F The type of the function.
     * @param task The function to push.
     */
    template <typename F>
    void push_task(const F &task)
    {
        tasks_waiting++;
        {
            const std::scoped_lock lock(queue_mutex);
            tasks.push(std::function<void()>(task));
        }
    }

    /**
     * @brief Push a function with arguments, but no return value, into the task
     * queue.
     * @details The function is wrapped inside a lambda in order to hide the
```

```
 * arguments, as the tasks in the queue must be of type
 * std::function<void()>, so they cannot have any arguments or return value.
 * If no arguments are provided, the other overload will be used, in order
 * to avoid the (slight) overhead of using a lambda.
 *
 * @tparam F The type of the function.
 * @tparam A The types of the arguments.
 * @param task The function to push.
 * @param args The arguments to pass to the function.
 */
template <typename F, typename... A>
void push_task(const F &task, const A &...args)
{
    push_task([task, args...] { task(args...); });
}

/**
 * @brief Reset the number of threads in the pool. Waits for all submitted
 * tasks to be completed, then destroys all threads and creates a new thread
 * pool with the new number of threads.
 *
 * @param _thread_count The number of threads to use. The default value is
 * the total number of hardware threads available, as reported by the
 * implementation. With a hyperthreaded CPU, this will be twice the number
 * of CPU cores. If the argument is zero, the default value will be used
 * instead.
 */
void reset(const ui32 &_thread_count = std::thread::hardware_concurrency())
{
    wait_for_tasks();
    running = false;
    destroy_threads();
    thread_count = _thread_count ? _thread_count :
        std::thread::hardware_concurrency();
    threads.reset(new std::thread[_thread_count ? _thread_count :
        std::thread::hardware_concurrency()]);
    running = true;
    create_threads();
}

/**
 * @brief Submit a function with zero or more arguments and no return value
 * into the task queue, and get an std::future<bool> that will be set to
 * true upon completion of the task.
 *
 * @tparam F The type of the function.
 * @tparam A The types of the zero or more arguments to pass to the
 * function.
 * @param task The function to submit.
 * @param args The zero or more arguments to pass to the function.
 * @return A future to be used later to check if the function has finished
 * its execution.
 */
```

```cpp
template <typename F, typename... A,
typename = std::enable_if_t<std::is_void_v<std::invoke_result_t<std::decay_t<F>,
    std::decay_t<A>...>>>>
std::future<bool> submit(const F &task, const A &...args)
{
    std::shared_ptr<std::promise<bool>> promise(new std::promise<bool>);
    std::future<bool> future = promise->get_future();
    push_task([task, args..., promise] {
        task(args...);
        promise->set_value(true);
    });
    return future;
}

/**
 * @brief Submit a function with zero or more arguments and a return value
 * into the task queue, and get a future for its eventual returned value.
 *
 * @tparam F The type of the function.
 * @tparam A The types of the zero or more arguments to pass to the
 * function.
 * @tparam R The return type of the function.
 * @param task The function to submit.
 * @param args The zero or more arguments to pass to the function.
 * @return A future to be used later to obtain the function's returned
 * value, waiting for it to finish its execution if needed.
 */
template <typename F, typename... A,
    typename R = std::invoke_result_t<std::decay_t<F>, std::decay_t<A>...>,
    typename = std::enable_if_t<!std::is_void_v<R>>>
std::future<R> submit(const F &task, const A &...args)
{
    std::shared_ptr<std::promise<R>> promise(new std::promise<R>);
    std::future<R> future = promise->get_future();
    push_task([task, args..., promise] {
        promise->set_value(task(args...));
    });
    return future;
}

/**
 * @brief Wait for all submitted tasks to be completed - both those that are
 * currently being executed by threads, and those that are still waiting in
 * the queue. To wait for a specific task, use submit() instead, and call
 * the wait() member function of the generated future.
 */
void wait_for_tasks()
{
    while (tasks_waiting != 0)
    {
        std::this_thread::yield();
    }
}
```

```cpp
    // ===========
    // Public data
    // ===========

    /**
     * @brief The duration, in microseconds, that the worker function should
     * sleep for when it cannot find any tasks in the queue. If set to 0, then
     * instead of sleeping, the worker function will execute
     * std::this_thread::yield() if there are no tasks in the queue. The default
     * value is 1000.
     */
    ui32 sleep_duration = 1000;

private:
    // ========================
    // Private member functions
    // ========================

    /**
     * @brief Create the threads in the pool and assign a worker to each thread.
     */
    void create_threads()
    {
        for (ui32 i = 0; i < thread_count; i++)
        {
            threads[i] = std::thread(&thread_pool::worker, this);
        }
    }

    /**
     * @brief Destroy the threads in the pool by joining them.
     */
    void destroy_threads()
    {
        for (ui32 i = 0; i < thread_count; i++)
        {
            threads[i].join();
        }
    }

    /**
     * @brief Try to pop a new task out of the queue.
     *
     * @param task A reference to the task. Will be populated with a function if
     * the queue is not empty.
     * @return true if a task was found, false if the queue is empty.
     */
    bool pop_task(std::function<void()> &task)
    {
        const std::scoped_lock lock(queue_mutex);
        if (tasks.empty())
            return false;
        else
```

```cpp
        {
            task = std::move(tasks.front());
            tasks.pop();
            return true;
        }
    }

    /**
     * @brief A worker function to be assigned to each thread in the pool.
     * Continuously pops tasks out of the queue and executes them, as long as
     * the atomic variable running is set to true.
     */
    void worker()
    {
        while (running)
        {
            std::function<void()> task;
            if (pop_task(task))
            {
                task();
                tasks_waiting--;
            }
            else
            {
                if (sleep_duration)
                    std::this_thread::sleep_for
                        (std::chrono::microseconds(sleep_duration));
                else
                    std::this_thread::yield();
            }
        }
    }

    // ============
    // Private data
    // ============

    /**
     * @brief An atomic variable indicating to the workers to keep running.
     */
    std::atomic<bool> running = true;

    /**
     * @brief An atomic variable to keep track of how many tasks are currently
     * waiting to finish - either still in the queue, or running in a thread.
     */
    std::atomic<ui32> tasks_waiting = 0;

    /**
     * @brief A mutex to synchronize access to the task queue by different
     * threads.
     */
    mutable std::mutex queue_mutex;
```

```
    /**
     * @brief A queue of tasks to be executed by the threads.
     */
    std::queue<std::function<void()>> tasks;

    /**
     * @brief The number of threads in the pool.
     */
    ui32 thread_count;

    /**
     * @brief A smart pointer to manage the memory allocated for the threads.
     */
    std::unique_ptr<std::thread[]> threads;
};
```

6.3 The `synced_stream` helper class

`synced_stream` is an optional helper class to synchronize printing to an output stream by different threads. It is not required for the thread pool to work.

```
class synced_stream
{
public:
    /**
     * @brief Construct a new synced stream.
     *
     * @param _out_stream The output stream to print to. The default value is
     * std::cout.
     */
    synced_stream(std::ostream &_out_stream = std::cout)
        : out_stream(_out_stream){};

    /**
     * @brief Print any number of items into the output stream. Ensures that no
     * other threads print to this stream simultaneously, as long as they all
     * exclusively use this synced_stream object to print.
     *
     * @tparam T The types of the items
     * @param items The items to print.
     */
    template <typename... T>
    void print(const T &...items)
    {
        const std::scoped_lock lock(stream_mutex);
        (out_stream << ... << items);
    }

    /**
     * @brief Print any number of items into the output stream, followed by a
     * newline character. Ensures that no other threads print to this stream
     * simultaneously, as long as they all exclusively use this synced_stream
     * object to print.
```

```
     *
     * @tparam T The types of the items
     * @param items The items to print.
     */
    template <typename... T>
    void println(const T &...items)
    {
        print(items..., '\n');
    }

private:
    /**
     * @brief A mutex to synchronize printing.
     */
    mutable std::mutex stream_mutex;

    /**
     * @brief The output stream to print to.
     */
    std::ostream &out_stream;
};
```

6.4 The `timer` helper class

`timer` is another optional helper class, which can be used to measure execution time for benchmarking purposes. It is not required for the thread pool to work.

```
class timer
{
    typedef std::int_fast64_t i64;

public:
    /**
     * @brief Start (or restart) measuring time.
     */
    void start()
    {
        start_time = std::chrono::steady_clock::now();
    }

    /**
     * @brief Stop measuring time and store the elapsed time since start().
     */
    void stop()
    {
        elapsed_time = std::chrono::steady_clock::now() - start_time;
    }

    /**
     * @brief Get the number of milliseconds that have elapsed between start()
     * and stop().
     *
     * @return The number of milliseconds.
     */
```

```
    i64 ms() const
    {
        return(std::chrono::duration_cast
            <std::chrono::milliseconds>(elapsed_time)).count();
    }

private:
    /**
     * @brief The time point when measuring started.
     */
    std::chrono::time_point<std::chrono::steady_clock>
        start_time = std::chrono::steady_clock::now();

    /**
     * @brief The duration that has elapsed between start() and stop().
     */
    std::chrono::duration<double>
        elapsed_time = std::chrono::duration<double>::zero();
};
```

References

[1] A. Williams, *C++ Concurrency in Action*. Manning Publications, 2019.

[2] B. Stroustrup, *The C++ Programming Language: The C++ Programm Lang_p4*. Pearson Education, 2013.

[3] B. Stroustrup, *Programming: Principles and Practice Using C++*. Pearson Education, 2014.

[4] B. Stroustrup, *A Tour of C++*. C++ In-Depth Series, Pearson Education, 2018.

[5] I. 14882:2017, *Programming languages — C++*. ISO, 2017.

[6] Microsoft Corporation, "<future> functions." 2016. [Online; accessed 2021-04-30] URL: https://docs.microsoft.com/en-us/cpp/standard-library/future-functions

[7] OpenMP Architecture Review Board, M. Klemm, and B. de Supinski, *OpenMP Application Programming Interface Specification Version 5.1*. 2020.

[8] S. D. Casey, "How to determine the effectiveness of hyper-threading technology with an application," *Intel Technology Journal*, vol. 6, no. 1 p.11, 2011.